Creating an IDE-independent Version of the Saros Whiteboard Using Web-Technologies

Abdullah Barhoum
Student ID: 5041774
abdoo2@zedat.fu-berlin.de

First Reviewer: Prof. Dr. Lutz Prechelt
Second Reviewer: Prof. Dr. Claudia Müller-Birn
Supervisor: Franz Zieris

Bachelor's thesis at the Software Engineering Research Group of the Institute of Computer Science at Freie Universität Berlin
August 30, 2018

Abstract:

Saros is an open-source Eclipse plug-in for distributed collaborative software development with a wide variety of features, including a Whiteboard for communicating topics graphically.

Many efforts have been made to achieve an Eclipse-independent implementation of Saros and all of its features. This thesis will contribute to this ongoing process by separating the logic of the Whiteboard from its Eclipse GUI while also preserving the current infrastructure, especially the implemented SXE synchronization protocol (Shared XML Editing).

Furthermore, this thesis will focus on the process of re-creating the Whiteboard's user interface with a more user-oriented approach and using Web-technologies such as HTML/ CSS/JavaScript which can run in web browsers cross-platform and cross-IDE seamlessly.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis oder in den Fußnoten angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Affirmation of independent work

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such.

Berlin, August 30, 2018
Abdullah Barhoum

Contents

1 Introduction

1.1 Pair Programming (PP)

Pair Programming describes the collaboration process between two developers using one computer to implement a specific feature, fix a bug, write unit tests or to simply review code; with the goal of achieving a higher-quality solution of the problem in hand.

A more profound definition of Pair Programming can be found in the book *Pair Programming Illuminated*:

> “Pair programming is a style of programming in which two programmers work side by side at one computer, continually collaborating on the same design, algorithm, code, or test. One of the pair, called the driver, is typing at the computer or writing down a design. The other partner, called the navigator, has many jobs, one of which is to observe the work of the driver, looking for tactical and strategic defects.” [15, p.3]

Pair Programming is the core building block of XP (eXtreme Programming), an agile software development method which became rather popular by the hands of K. Beck in his book: *Extreme Programming Explained: Embrace Change.*

1.2 Remote pair programming (RPP)

Reviewing the definition of pair programming from the first section, it is directly stated that both programmers need to work on one computer.

However, the rapid development of internet services, fast connections and international businesses among other factors forced a new definition of pair programming in the present time and introduced the concept of Remote Pair Programming, where the driver and the observer do not need to be in the same room to achieve the benefits of traditional pair programming.

Many tools have been developed to help practice remote pair programming, including:

- Screen Sharing Tools:
  These tools are based on the WYSIWIS abstraction (What You See Is What I See - pronounced “whizzy whiz”). [14]
  With this type of tools, one of the developer’s desktop is transferred to the other one so that both participants see the same editor contents and editing activities as if they were in the same room.
  And for interacting with the computer, either all input device streams are merged into a single input stream or, more commonly, strict floor control is used which will allow input only by one participant at any time. [14]

- Cloud development environments or cloud IDE[1]s:
  Web-based IDEs, where the files are stored on a server with a fully-featured

[1]Integrated Development Environment

compiler, debugger and a GUI[2], most (if not all) of the features can be accessed easily with a standard web browser, which makes it very portable.

- IDE plug-ins:
  special pieces of software that can be installed on IDEs to mirror changes on files or projects in real time to all participants.
  Combined with a voice chat service (mainly VoIP[3]) these plug-ins allow remote pair programming with minimal sacrifices to each developer's setup and comfort.
  One of these plug-ins is Saros, which is in the focus of this thesis.

1.3 Saros

Saros is an open-source IDE plug-in for distributed collaborative software development, it takes the concept of remote pair programming a step further by allowing up to 5 participants in the session.

Saros started as a research project in the software engineering research group at Freie Universität Berlin, it became after that open-source and is further developed mainly by students at the university.

Saros includes a variety of features but focuses mainly on the real-time collaborative editor, which maps code changes to other participants in real-time so that all project instances remain consistent and in-sync, it even allows simultaneous editing of the same files by two or more users.

Currently, Saros is only available for the Eclipse IDE, however, an IntelliJ version is in development.

1.4 Saros Development Process

First I am going to first explain a couple of concepts and tools related to the development of Saros, and then connect the dots to get the full picture.

Version Control System (VCS): is a system that records changes to a file or set of files over time so that it is possible to recall specific versions later. A common use of VCS is monitoring the changes of software source code, though, in reality, it is possible to do this with nearly any type of file on a computer. One of the most known version control systems is Git.

Git: a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency[4]. The current state of the project is usually saved in the project's repository on the 'master' branch. At any time a developer can make a copy of the master onto their own machine and start making changes or adding new features. Once done the developer can finally integrate their changes into the repository by committing and pushing them into the master.

Continuous Integration (CI): as defined by *Fowler* [8]: "[CI] is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per

[2]Graphical User Interface
[3]Voice over IP
[4]https://git-scm.com/

day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."

Code Review or Peer Review: is the practice of having team members critique changes to a software system. It is intended to find mistakes overlooked in software development, improving the overall quality of software. It is worth noting here that the scope of the changes is a critical factor since a huge amount of changes will be intimidating to review. The Saros's development guide recommends that patches should not take more than 15 minutes to be reviewed[5].

Gerrit: "is a modern code review tool that facilitates a traceable code review process for Git-based software projects. Gerrit tightly integrates with test automation and code integration tools. Authors upload patches, i.e., collections of proposed changes to a software system, to a Gerrit server and await reviews" [12].

By the time of writing this thesis, the Saros team has switched from using Gerrit to Github Pull requests, but they both serve the same purpose in the development process.

The Saros development process uses all of the above-mentioned concepts and tools; to make it clear I will give an example of adding a new feature to Saros:

The developer uses Git to check out the latest version of the system from the master branch and starts implementing the new feature locally. when done, the developer uploads a patch to Gerrit, which creates a new review request and triggers some CI tools that build and test the patch. In case the reviewers or the CI tools find any errors or problems, the developer uploads another patch addressing these problems, and like the first patch it will be tested and put to review again. If the patch does not cause any build or test errors and the reviewers agree with the idea and logic behind it, it will be submitted and then merged into the master branch of the repository.

1.5 Saros Whiteboard

Even though writing code is an essential step of software development, one should not underestimate the importance of software design; a well thought-out software architecture guarantees high-quality code and eases solving possible algorithmic problems.

Traditional whiteboards play a big role in the software design process, they allow participants to share their mental models and own thoughts about the software in the shape of sketches or diagrams, which makes them easier to explain and to discuss with other participants, it is also possible to change or expand these diagrams in case any inconsistencies are found.

The importance of a whiteboard being included in Saros was clear in the surveys of Riad Djemili, the creator of Saros, in 2006 [7]; one of the questions explicitly discussed a whiteboard feature:

[5]http://www.saros-project.org/review

> You have a whiteboard editor which allows you to draw geometric shapes and diagrams which your partner can see

Examining the results: 45% of the participants found this feature “nice to have”, while 27% found it to be important and 9% answered with *essential*. Only 11% refused the idea.

the Saros Whiteboard was first prototyped for Eclipse IDE by Michael Jurke in 2010 [10], the focus of Jurke’s work was to answer all fundamental questions related to the development of the Whiteboard, the main focus being the synchronization method.

The Whiteboard was later expanded by Hendrik Degener in 2012 [6] with focus on usability, user experience, and graphic-tablets support. He added new features such as:

- Draw diamonds
- Draw arrows with different head types and line styles, as well as ”magnetic” arrows, which are connected to other shapes and remain connected if these change position or size.
- Color the shapes
- Highlight specific shapes to draw the attention of other users
- Save the session and restore it later
- Export the Whiteboard’s contents as an image

While the prototype of Jurke was fully integrated into Saros, the work of Degener, however, did not, because the patch was too big[6]. With almost 20,000 lines it was impossible to review in one go. (See 1.4)

The Saros team has then tried to divide the patch into much smaller parts and integrate them, this process was however abandoned by the end of 2014, thus resulting in the work of Degener not being adopted into Saros, and the Whiteboard remained at its original state.

1.6 Motivation

Many efforts have been made to achieve an Eclipse-independent implementation of Saros and all of its features with the goal of reducing code duplication, complexity and porting cost to other IDEs and/or environments. The rise of web technologies in the past few decades offered the needed independency.

Web technologies are built upon the idea of providing means of communication between different platforms, devices, systems and even users. An example being websites, which can range from a simple static portfolio to a fully immersive 3D game, the only requirement being a web browser.

It is worth noting here that using external libraries in the development environment of web applications does not mean any added dependencies to the runtime

[6]https://saros-build.imp.fu-berlin.de/gerrit/#/c/595/

environment; web technologies are built upon browsers being the target environment, so any external libraries used will be embedded in the source code, guaranteeing their availability.

Christian Cikryt has done a very extensive evaluation on the use of a web browser for unifying GUI development across IDEs [5]. In his master's thesis he discussed the available browsers for Eclipse and IntelliJ, their advantages and disadvantages, and came to the conclusion that Saros should use the SWT[7] browser and that using web technologies is a promising solution for building an IDE-independent GUI, which is currently in development.

Consequently, this thesis will contribute to this development process by creating a web-based Whiteboard for Saros.

1.7 Goals

The main purpose of this thesis is to achieve an IDE-independent implementation of the Saros Whiteboard, which can be divided into sub-goals:

- Find and resolve entanglements between the Whiteboard's core logic and Eclipse, without sacrificing functionality and while also preserving as much of the available code as possible.

- Create a new IDE-independent GUI for the Whiteboard with all of the features available in the original Whiteboard, and if possible add new ones, taking the work of Degener [6] as a reference for new features and a more user-oriented approach.

1.8 Scope

As I first started working on this thesis, the scope was not clear. I knew that at least the GUI of the Whiteboard had to be recreated, but I did not know how many changes in the logic and the infrastructure are required to guarantee compatibility with the new GUI.

By skimming through the code quickly, it was clear that the network layer of the Whiteboard, responsible for sending and receiving messages, was completely independent of Eclipse, and thus will neither be altered nor discussed. After further analysis discussed in Section 2, the scope was limited to only re-implementing the GUI with very minimal changes in the underlying logic.

1.9 Outline

In Section 2, I analyze the code of the Whiteboard to identify the aforementioned entanglements and to find out how much of the whiteboard has to be re-implemented.

In Sections 3, 4 and 5, I discuss my attempts at implementing a prototype of the new Whiteboard, the problems faced, the decisions made and the reasoning behind it.

[7]The Standard Widget Toolkit

Finally, Section 6 contains final thoughts and an outlook of the work achieved in this thesis.

1.10 Methods

While trying to understand the Whiteboard's infrastructure, I referred to the work of the creator of the Whiteboard [10] as well as any follow-ups [6], I also created diagrams describing the structure of the Whiteboard and focusing specifically on inheritance hierarchies because of the highly abstracted code. To understand how the different components interacted with one another, I changed some parts of the Whiteboard, added logging statements and observed the resulting behavior.

In the implementation phase I started by creating quick-and-dirty prototypes to answer all fundamental questions, after that I focused on an iterative approach while adding or testing new features, taking into consideration the possibility of future development, developer-friendliness as well as user-friendliness.

2 Understanding the Whiteboard

In this section I will start by describing how the end-user will perceive and interact with the Whiteboard, then I will do a quick analysis of the Whiteboard's infrastructure, explain some concepts related to its building blocks and try to define a clear separation between the logic and the GUI of the Whiteboard for each of these blocks.

Please note that this section relies **heavily** upon the work of Micheal Jurke [10], refer to his diploma thesis for more detailed information about the development process and the decisions made.

Starting with the graphical components, the current Saros Whiteboard (See figure 1) is divided into three parts, The Toolbar in the top right corner, the Palette menu on the left side, and the drawing space which fills out the rest of the window.

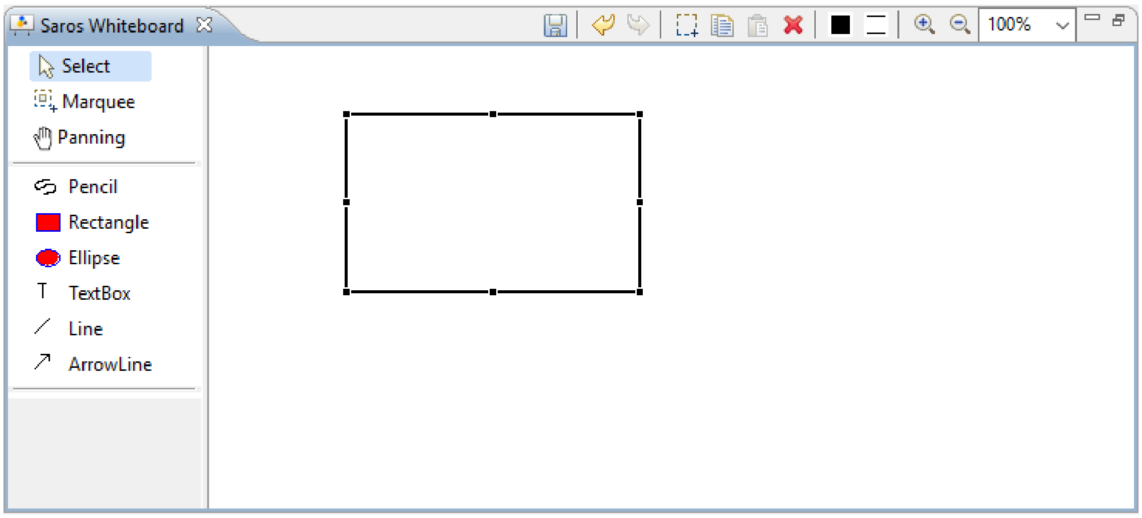


Figure 1: Screenshot of Jurke's Whiteboard

There are many tools on the Palette, with which it is possible to draw rectangles, ellipses, lines, and arrows. These tools use the same mechanism for creating the shapes: The user presses and holds the left mouse button on the drawing space, this mouse position defines one of the corners of the shape, and then by moving the mouse the second corner follows the mouse cursor, thus resizing the shape. The drawing process ends when the mouse button is released.

The pencil tool is used for freehand drawing, the drawing mechanism is similar to the tools mentioned above, the first mouse press defines the start of the line and every time the mouse moves a new point is added to this line. Another tool which has a slightly different behavior is the insert text tool, it functions by clicking on any position on the drawing space, a pop-up appears asking the user to input the wanted text, the text is then inserted at the click position.

The Whiteboard also has a selection tool, which allows the scaling and moving of the created shapes, it is also possible to delete a selected shape(s) by clicking the delete button from the Toolbar. The Toolbar contains additionally a collection of commands such as zoom, undo, redo, copy, paste and select all, there is also a button for exporting the contents as an image, but the feature itself is not implemented.

After any user interaction that affects the contents of the Whiteboard (e.g.: drawing a shape, resizing, removing, etc.), the changes to the state of the Whiteboard are synchronized with all other users in the same session.

Now to understand how the Whiteboard works internally, I am going to explain some concepts first.

2.1 DOM: Document Object Model

The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure, and style of documents [1].

To make it more clear, here is an example HTML document:

Listing 1: Example HTML Document

```
<html>
  <head>
    <title> Example </title>
  </head>
  <body>
    <h1> This is a header </h1>
    <div>
      <h2> This is another header </h2>
    </div>
  </body>
</html>
```

And the equivalent DOM tree would be:

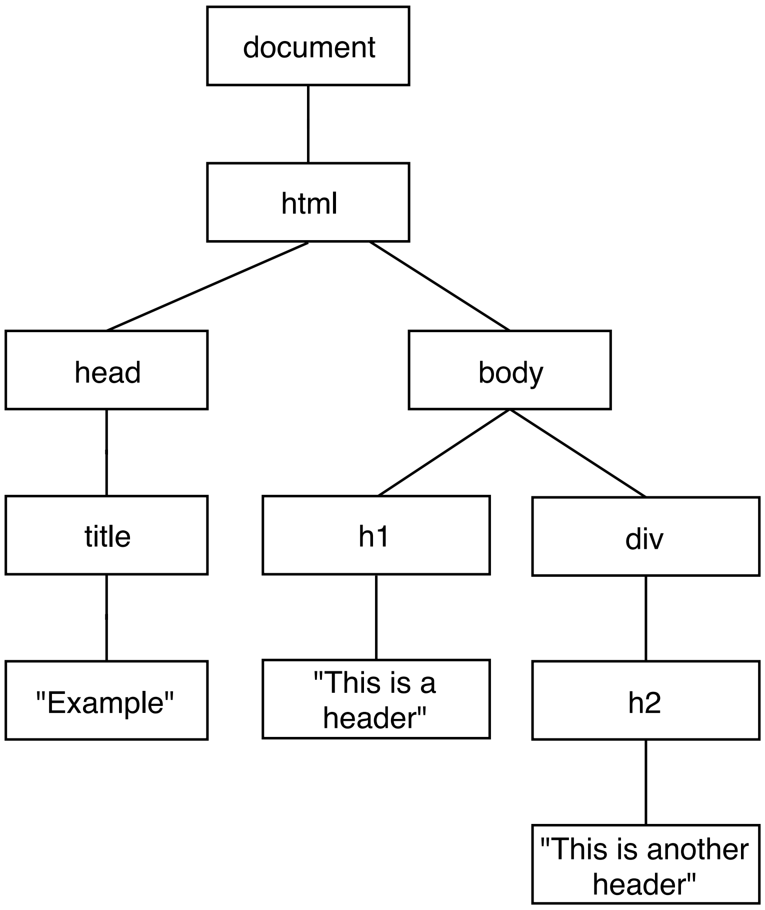


Figure 2: Example HTML-DOM tree

With DOM access, manipulating the contents of the page becomes trivial. For example, to change the title of the document, the following JavaScript code snippet should be executed:

```
let html = document.children[0];
let head = html.children[0];
let title = head.children[0];
title.innerText = "Modified";
```

The structure of the Whiteboard's DOM is similar to the HTML document above, the nodes, in this case, describe the different shapes on the Whiteboard as well as their dimensions and attributes.

As an example, figure 3 is the DOM tree of the Whiteboard containing a rectangle and an ellipse.

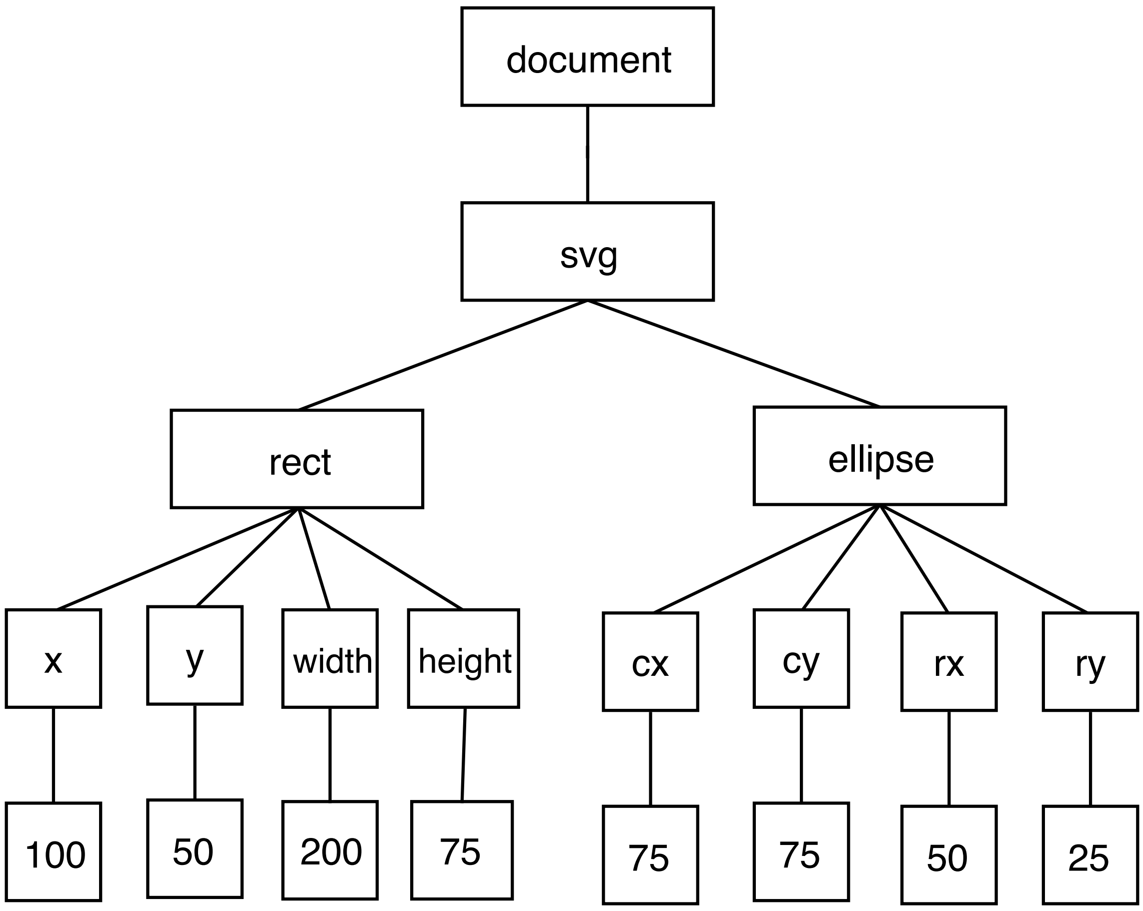


Figure 3: Example Whiteboard DOM tree

The DOM of the Whiteboard always has a root element of type SVG, even when empty.

Since all contents of the Whiteboard are DOM objects in respect to SXE `Records` (will be explained in the next section), they can be easily modified.

2.2 SXE: Shared XML Editing

Firstly, what is XML?

2.2.1 eXtensible Markup Language (XML)

XML is a simple, very flexible text format, originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.[8]

XML is a software- and hardware-independent tool for storing and transporting data, it was designed to be self-descriptive and both human- and machine-readable.[9]

[8] `https://www.w3.org/XML/`
[9] `https://www.w3schools.com/xml/xml_whatis.asp`

Notice the simplicity in the following example:

Listing 2: Example XML

```
<note date="12/07/2018">
  <to>Bob</to>
  <from>Alice</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

2.2.2 The Shared XML Editing Protocol (SXE)

SXE is a protocol for collaboratively editing XML files. As explained by Jurke [10]: The base principle of SXE can be described by the term "record". Records correspond to the data structure of the shared artifact and are also means of communication. There is no way to change the shared artifact except through records.

The SXE protocol provides a simple way of synchronizing an unordered set of records across several endpoints. If we imagine the Whiteboard as an XML document shared between the participants, then all changes to the Whiteboard are just changes to this document and thus can be synchronized with SXE.

Jurke has implemented the essential parts of SXE, but not everything, mainly because other features are not needed in the Whiteboard, consequently, the explanation in this section touches upon only what is implemented.[10]

I am going to explain SXE records using a running example, starting by the state of the Whiteboard when its empty.

Listing 3: XML representation of the Whiteboard's state when empty

```
<svg xmlns="http://www.w3.org/2000/svg">
</svg>
```

SXE has three different record types, which describe modifications to an XML document, these types are:

- The NEW record:
  corresponds to new DOM nodes and their specializations respective the type field. NEW records basically create new DOM nodes using the information they contain.
  For example, to create a rectangle with `x: 80, y: 150, width: 120, height: 200`, the following NEW records have to be applied:

  Listing 4: Example NEW records

  ```
  <new name="rect" rid="rectangel" type="element" version="0"
  parent="root" primary-weight="0.16"/>

  <new name="x" rid="attributeX" type="attr" version="0"
  parent="rectangel" chdata="80" primary-weight="0.92"/>

  ```

[10]Please refer to the original specification of SXE for more information: https://xmpp.org/extensions/xep-0284.html

```
<new name="y" rid="attributeY" type="attr" version="0"
parent="rectange1" chdata="150" primary-weight="1.38"/>

<new name="width" rid="attributeWidth" type="attr" version="0"
parent="rectange1" chdata="120" primary-weight="2.58"/>

<new name="height" rid="attributeHeight" type="attr" version="0"
parent="rectange1" chdata="200" primary-weight="3.73"/>
```

These records contain *Fields* which hold important information, some of which are:

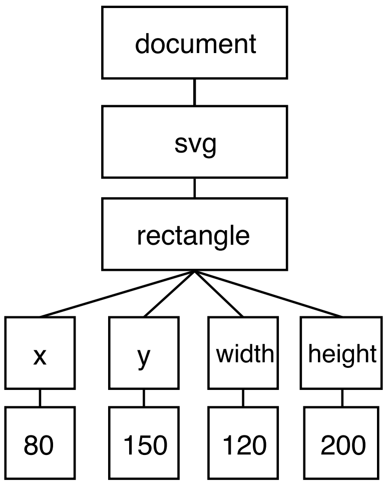
– rid: an ID, used to identify records across the document.
– type: the type of the DOM node, can be either 'element' or 'attr' (attribute).
– name: the name of DOM node (e.g.: 'ellipse' or 'width').
– chdata: only in nodes with type 'attr', contains the value of the attribute.

Other fields are used mainly for synchronization and conflict resolution.

As may be noticed, a record has to be created for every node of the DOM tree. Assuming the root `svg` element in the Whiteboard has the ID `rid="root"`, applying the records in the example above would result in a state that can be represented in XML as follows:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="80" y="150 width="120" height="200"/>
</svg>
```

The equivalent Whiteboard DOM tree would be:



• The SET record:
is used to apply changes to records and corresponds to the local history of changes to a DOM node. On conflicting edits, they can be used to discard changes and revert the state to a previous non-conflicting version.
To change the width of the rectangle from the previous example to 500, the following SET record is used:

Listing 5: Example SET record

```
<set target="attributeWidth" version="1" chdata="500"/>
```

resulting in the following XML representation:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="80" y="150 width="500" height="200"/>
</svg>
```

- The REMOVE record:
  is used to delete nodes, and comes along with a requirement: clients have to apply REMOVE records for all children nodes before removing the desired node.
  To remove the rectangle we created earlier, we have to apply REMOVE records for all of the attributes first, then for the rectangle itself.

  Listing 6: Example REMOVE records

  ```
  <remove target="attributeX"/>
  <remove target="attributeY"/>
  <remove target="attributeWidth"/>
  <remove target="attributeHeight"/>

  <remove target="rectange1"/>
  ```

  returning the Whiteboard to its original state:

  ```
  <svg xmlns="http://www.w3.org/2000/svg">
  </svg>
  ```

  Note: because of performance issues when trying to remove a lot of elements in one go; REMOVE records are not implemented, instead SET records that change the visibility of the target to false are used, thus having the same effect. While it is arguable that this is a problem since it causes memory leaking, the effects are rather minimal since all records are deleted when a new session starts.

2.3 The Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object (called the Subject) changes state, all its dependents (called the Observers) are notified and updated automatically [9]. The problem that the Observer pattern solves is how to maintain consistency among these objects in a way that promotes reuse, keeping a low coupling among classes.

The change of state of the Subject is called an *event*, and the Observers are notified of these events by calling one of their functions, usually called the *handler*.

2.4 MVC: Model-View-Controller

MVC programming is the factoring of application components, whereby objects of different classes take over the operations related to the application domain (the

Model), the display of the application's state (the View), and the user interaction with the model and the view (the Controller) [11].

The interactions between the different components can be described by the following illustration:

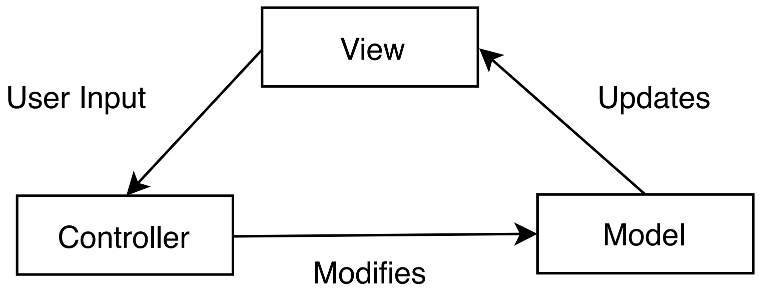


Figure 4: Interactions between the Model, the View and the Controller in MVC

I will now briefly describe each component of MVC and its respective counterpart in the Whiteboard.

2.4.1 The Model

Definition:

The model of an application is the domain-specific software simulation or implementation of the application's central structure. This can be as simple as an integer or a string, or it can be a complex object [11].

The model is built without any knowledge about views and controllers. There is an implicit association via an observer mechanism. The model must implement the functionality defined in the role 'Subject' and introduce notifications in all places where the state of the model is changed.

Whiteboard context:

The models in the Whiteboard are called `Records`, which correspond to SXE records. These `Records` represent the state of the Whiteboard with its contents, and are divided into `NodeRecords` and `SetRecords`.

Each `NodeRecord` is a node in the DOM tree of the Whiteboard and corresponds to SXE NEW record, and is in turn divided into two types:

- `ElementRecords`: equivalent to NEW records where the type field is set to 'element', they contain information about the type of a shape (for example a circle or ellipse) and can have children `ElementRecords` or `AttributeRecords`.

- `AttributeRecords`: equivalent to NEW records where the type field is set to 'attr', and contain information about the dimensions and attributes of the parent `ElementRecord` (for example x = 50 or color = 0,0,0) and cannot have children of any type.

`SetRecords` correspond to SXE SET records and represent changes done on a `NodeRecord`. Since the list of `SetRecords` for a specific element represents the history of this element, they are used to restore the target `NodeRecord` to a previous state in case of a conflict.

Note: SXE REMOVE records have no equivalent in the Model, instead `SetRecords` that change the `visibility` to `false` are used.

All of the aforementioned classes are GUI- and Eclipse-independent and can be used in the new Whiteboard. The GUI related part of the Model starts by the abstract class `LayoutElementRecord`, which delivers information such as *element type, x, y, width, and height* used by the GEF (See 2.4.2) to render the actual graphical element on the drawing space.

The structure of the Whiteboard's Model as of the writing of this thesis is as follows:

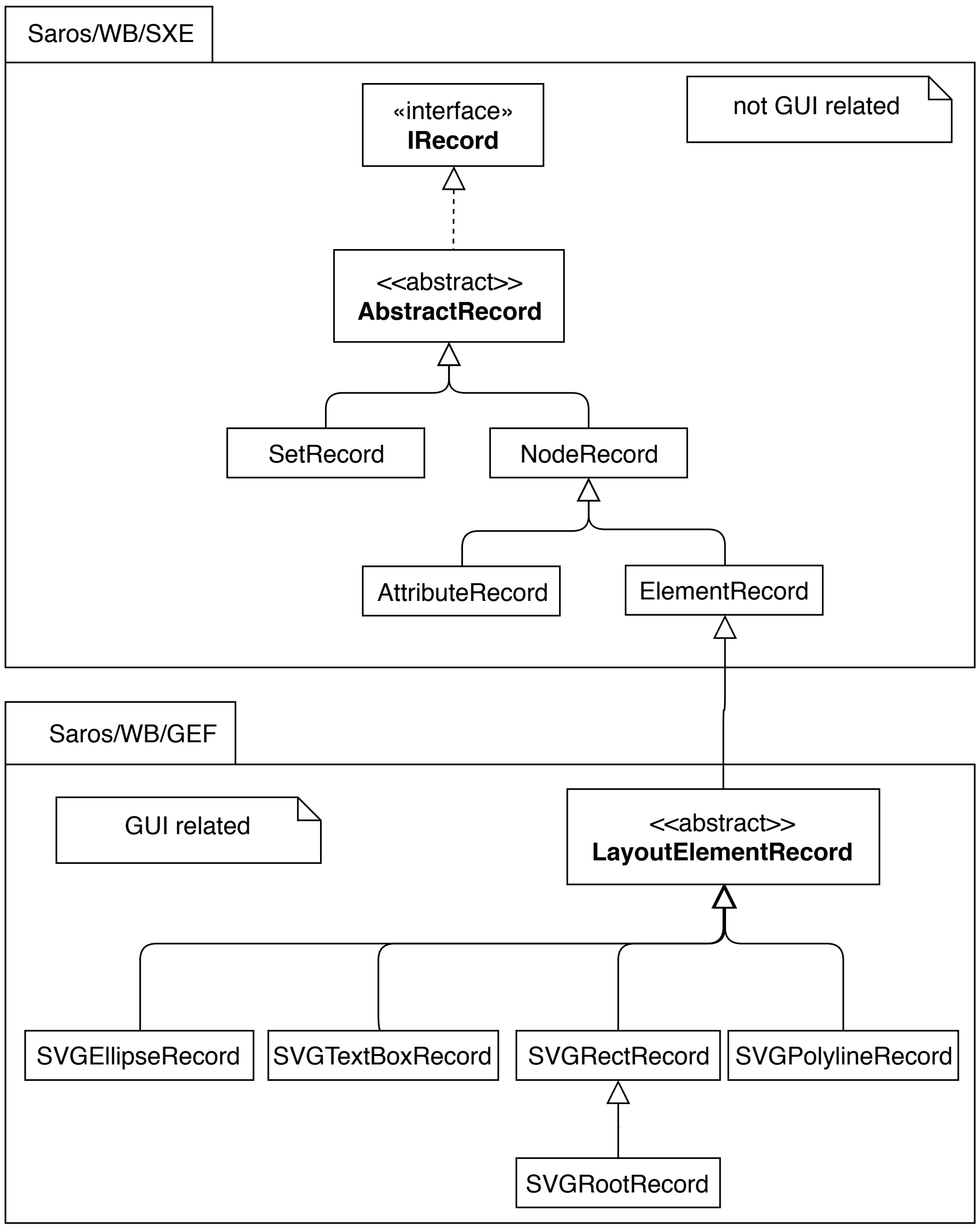


Figure 5: UML Class diagram: Records inheritance hierarchy

2.4.2 The View

Definition:

Views deal with everything graphical; they are responsible for displaying the application's state using the data they get from the models. They contain not only the components needed for displaying the data but can also contain subviews and be contained within superviews [11].

Multiple views can be attached to one model, where the views among themselves can differ. This is not surprising because there are many possibilities to represent the same set of data.

Whiteboard context:

The view is completely built using GEF:

Eclipse's Graphical Editing Framework (GEF): is an Eclipse framework used mainly as a foundation for GUI builders. It has a history of 17 years active development and is a stable and actively supported open source feature for Eclipse.

It allows modifying models graphically so that the user can directly interact with the model by a figure. GEF handles the mouse and keyboard events and interprets the inputs and key bindings. The goal is that developers do not need to re-invent the wheel for generally necessary functions.

In the context of the Whiteboard, GEF provides the GUI of the Whiteboard using `GraphicalEditorWithPalette`, which creates an editor with one side containing a `PaletteViewer` (to displays the available tools), and the other a `GraphicalViewer` (to display and interact with figures). GEF also provides the views for the models of the Whiteboard, these views are called `EditParts`, which are the rectangles, ellipses, lines, etc.. that the user sees on the drawing space of the Whiteboard. Each `LayoutElementRecord` has a corresponding `EditPart` as a view.

The GEF uses the Command design pattern, each command corresponds to a user-interaction and provides an execute, undo and redo functionality. The current Whiteboard has all essential commands already implemented:

- `CopyRecordCommand`:
  is used when copying a selected shape to paste it later, the command actually copies the `ElementRecord` corresponding to the selected shape. Note that this command does not affect the model at all, that is why it is completely separated from SXE.

- `PasteRecordCommand`:
  is used to paste the copied shapes from the `CopyRecordCommand`, this command also makes sure that the duplicated elements have slightly adjusted positions to make it clear to the user that the shapes have indeed been duplicated. Without these adjustments, the new shapes will be created directly on top of the old ones, as if nothing changed.

- Instances of `AbstractElementRecordCreateCommand`:
  these are the commands used when creating the shapes after the drawing process is done.

- `ElementRecordChangeLayoutCommand`:
  this command changes the position and size of a shape by changing the values in the corresponding `AttributeRecords`.
- `ElementRecordAddCommand`:
  is used to remove an `ElementRecord` from its parent and append it as a last child to another one.
- `DeleteRecordsCommand`:
  this command allows the deletion of an arbitrary group of `ElementRecords` by only removing the top-most elements.

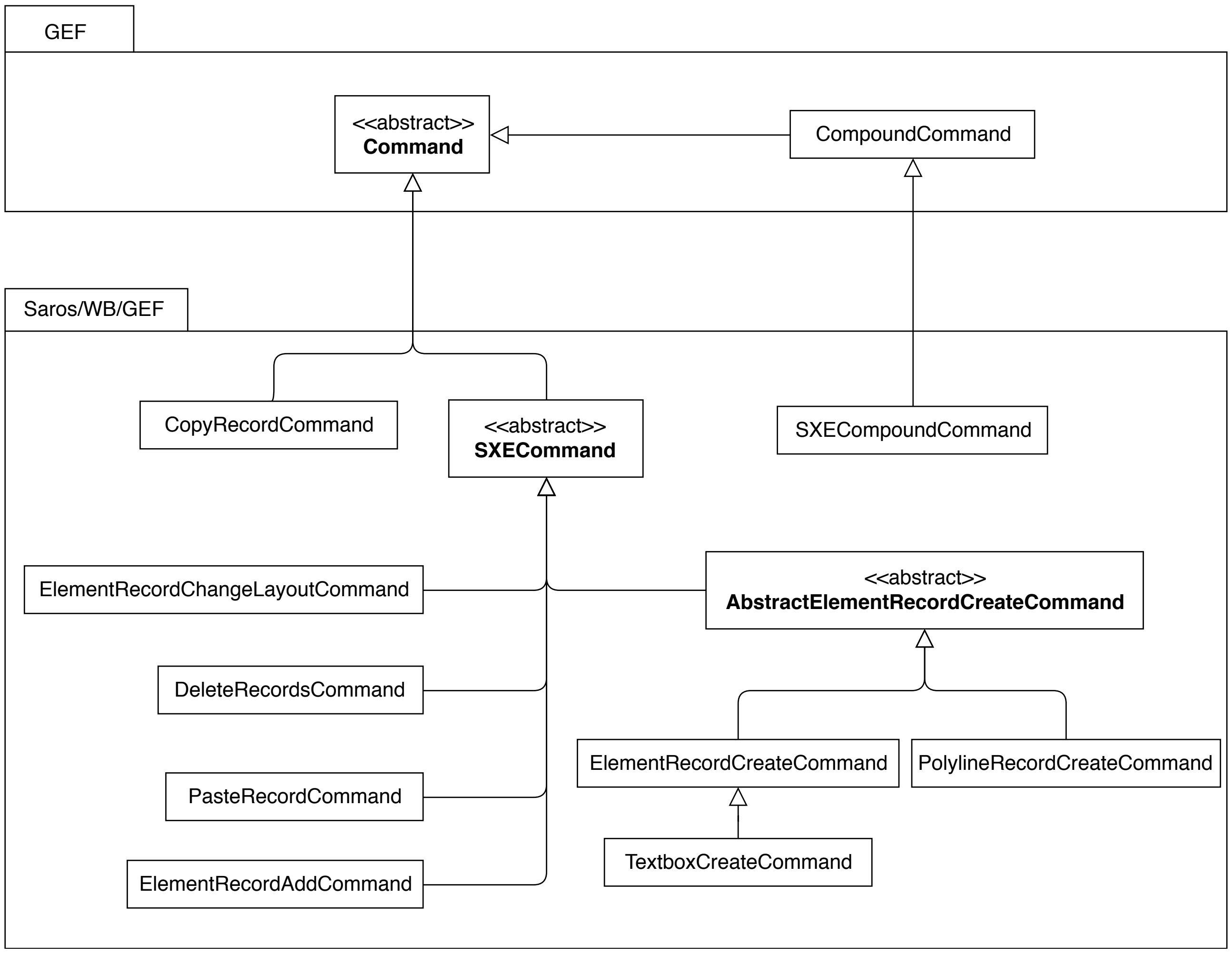


Figure 6: UML Class diagram: Available commands in the Whiteboard

Notice that in figure 6, all commands except the `CopyRecordCommand` extend the abstract `SXECommand` class, that is because these commands affect the model, and these changes need to be synchronized with other Saros users.

Unfortunately, these commands are based on the GEF's command and cannot be reused out of Eclipse, and since the GEF is programmed for Eclipse, it has to be replaced in the new Whiteboard, thus a re-creation of the GUI is needed.

2.4.3 The Controller

Definition:

Controllers are responsible for the management of views and the corresponding model, they also provide the interface between the model and the interactive user interface devices, as if they were 'translating' from GUI events to application logic [11].

Whiteboard context:

The Whiteboard has a one and central `SXEController`, it handles user interactions and reflects them as `Records`. All of the above mentioned Commands create `Records` that are applied internally by the `SXEController`.

If record application was locally successful, the `Records` will be sent to peers using the `SXETransmitter`, and any invalid `Records` will be discarded. An example of an invalid `Record` would be a one that creates a circular relationship between two `ElementRecords`; as in each of the `ElementRecords` is a parent and a child of the other one at the same time.

The `SXEController` is also responsible for the application of remote `Records` and the resolving of any conflicts that might result due to synchronization problems.

The controller is completely independent of Eclipse and can be used in the new Whiteboard.

2.5 Conclusion

It became clear after this analysis that the scope of this thesis would be limited to a new implementation of the Whiteboard's GUI with minimal to no changes to the actual logic.

3 The First Prototype

Since I did not have any experience implementing drawing applications, I decided to start with a quick-and-dirty prototype to get to know the domain of such software, the type of problems I will be facing as well as get familiar with the tools I will be using. I also decided to start with a standalone version of the Whiteboard that works out of Eclipse to simplify the development and the testing.

Soon enough, I was faced with the first problem: how should shapes be displayed on-screen?

3.1 Finding the Best Rendering Engine

Two of the technologies that are rather popular and directly come to mind are **Scalable Vector Graphics (SVG)** and HTML **Canvas**, other technologies include **Cascading Style Sheets (CSS)** and **Flash**. I decided however to explore some Whiteboards that are available online and find out which technologies they used and how they worked. In the next section, I will go over the ones that I think stand out in comparison to the others.

3.1.1 Related Work

Realtime Board[11] is a professional collaborative Whiteboard. The drawing area stretches across the whole screen with some overlaying toolbars scattered nicely around the screen.

Realtime Board offers a lot: it allows users to draw simple shapes (rectangles, arrows, circles, ellipses, stars...) as well as change the shapes' colors, insert text into them, change their background color, change border-colors and thicknesses, change fonts, insert notes, move, resize, rotate and delete all of the aforementioned objects and many other features.

Realtime Board also has a free drawing mode with free choice of color and pen thickness, it also supports 'smart drawing mode', which allows users to create, manipulate and delete elements by drawing special gestures on the whiteboard.

Users have unlimited drawing space with the ability to zoom in and out, it is also possible to fit the view to all elements of the whiteboard. Adding images is supported, users can upload them from the local machine or request them directly from the internet. The whiteboard also offers a big variety of templates including flowcharts, mind maps and diagrams.

Realtime Board provides undo and redo ability with a history of all the actions done by all users, a chat feature as well as a voting feature, it also allows the user to export the contents as an image, PDF or CSV.

All of these features and tools are provided in a neat, compact and user-friendly interface. A very impressive piece of work indeed.

Realtime Board uses **Canvas** internally to render the whiteboard, while all the menus and button are standard HTML elements, but because this board a commercial software, I was not able to know more about how it was implemented or if any open-source libraries were used.

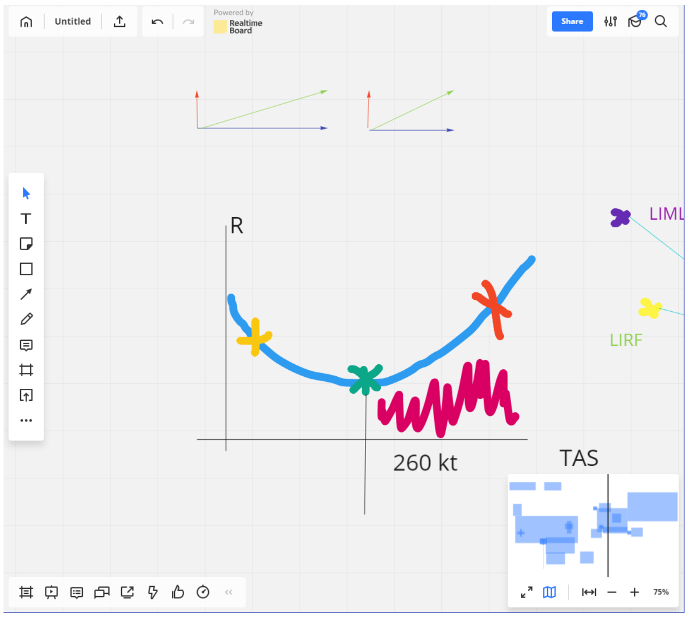


Figure 7: Screenshot of Realtime Board

[11] https://realtimeboard.com/

Sketchboard[12] is a whiteboard built with focus on creating diagrams. While it is possible to draw freely, the main method to create shapes is to drag them from the right-side menu to the drawing space.

It has a lot of pre-created shapes to choose from regarding many fields including software design, it also allows the users to connect the different shapes together with arrows. Many different arrow types are available, with useful customizations such as arrow thickness, the ability to flip the arrow or switch arrow lines between straight and curvy.

Users can use Markdown[13] to style text elements as they please.

The shapes are also customizable, users can change the position, the color, the size, the text alignment as well as the order (which shape is in front of or behind the other one) of each shape or drawing.

Furthermore, features such as unlimited drawing space, undo, redo, zoom and chat are also available.

Sketchboard uses **SVG** to render the shapes on-screen, this rendering is done using an open-source library called dojox.gfx which is a part of an enormous toolkit called Dojo[14].

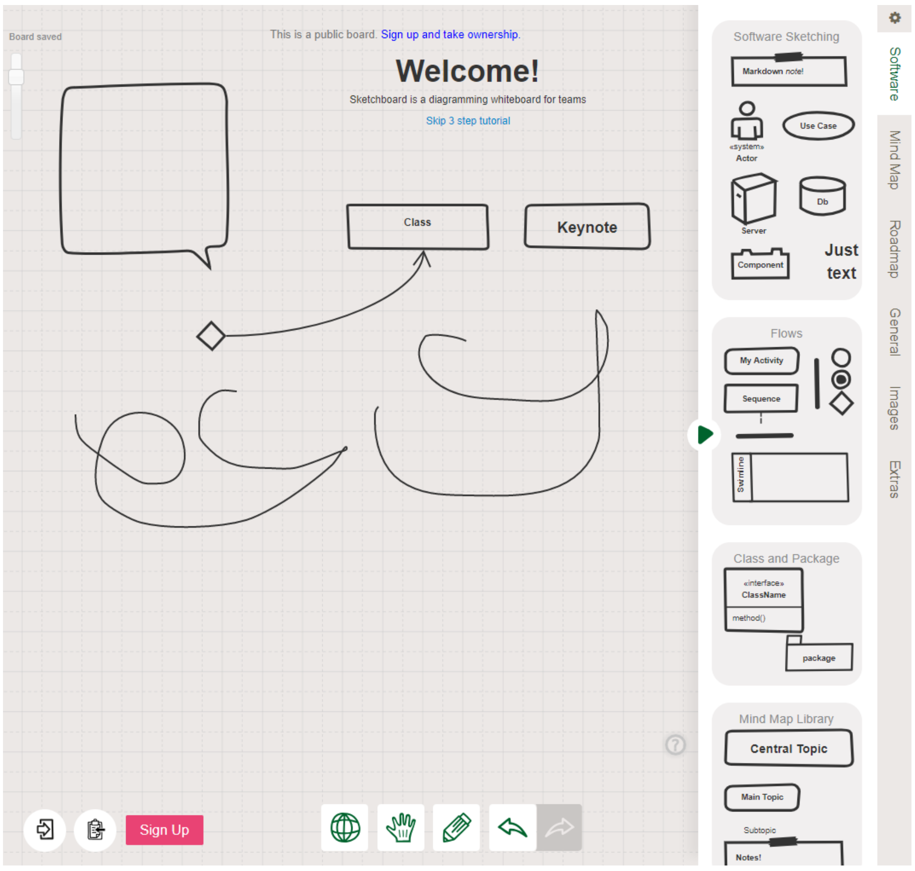


Figure 8: Screenshot of Sketchboard

[12]https://sketchboard.me/
[13]https://en.wikipedia.org/wiki/Markdown
[14]https://dojotoolkit.org/

AWW App[15] is an online whiteboard for real-time visual collaboration. It allows users to draw rectangles and circles, insert text and notes or use the mouse to freely draw different shapes.

It is possible to move and delete the shapes, but resizing is not available. The undo feature is implemented, redo however is not.

AWW App provides an unlimited drawing space, with the ability to export the contents as an image or PDF.

What makes this whiteboard shine is that it is embeddable and provides an API for developers to use it in their own websites.

AWW App uses **Canvas** for rendering.

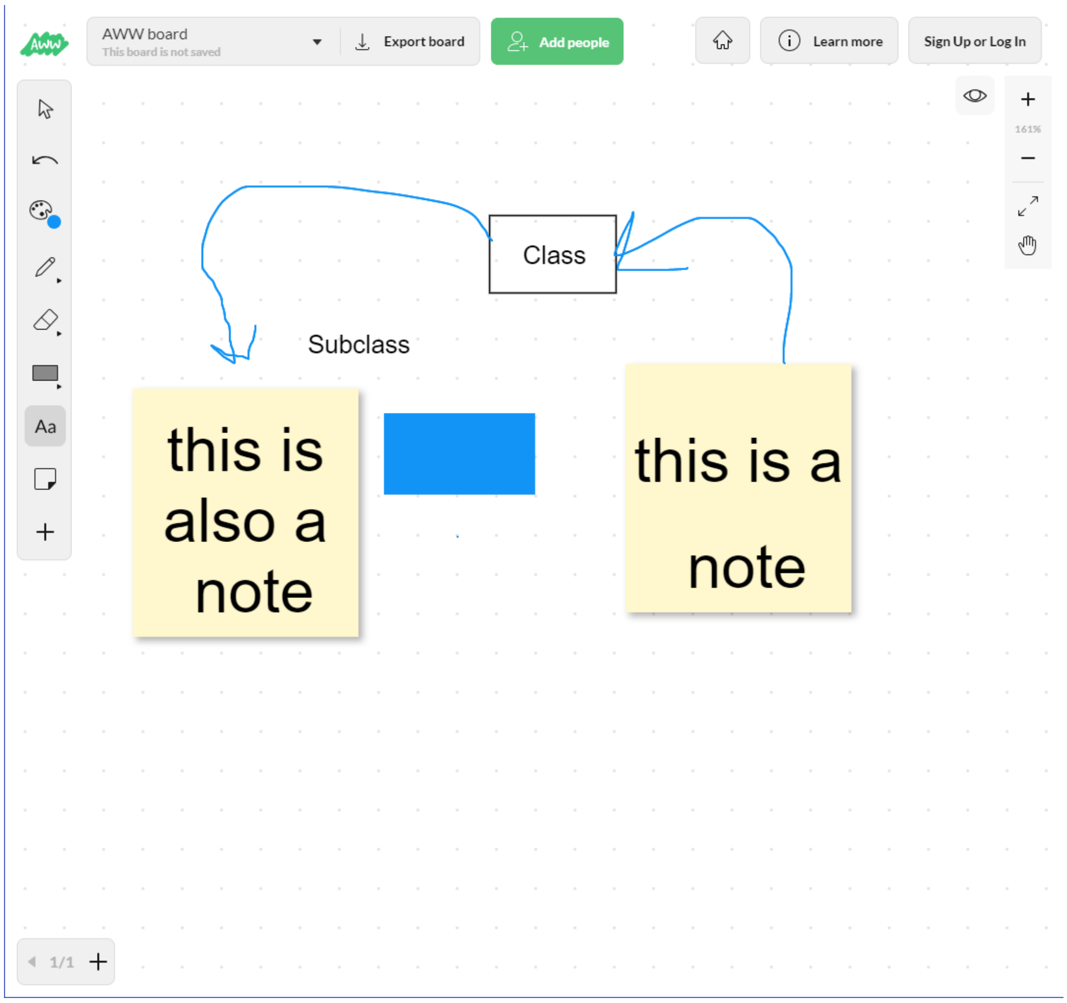


Figure 9: Screenshot of AWW App

Web Whiteboard[16] is a rather minimalistic online whiteboard, it has a very limited set of tools in the left-side menu: freehand drawing with 4 colors, undo and redo, text or note insertion and the ability to export the whiteboard's contents.

Resizing, moving, rotating and such are not available.

[15]https://awwapp.com/
[16]https://webwhiteboard.com/

Web Whiteboard uses **Canvas**.

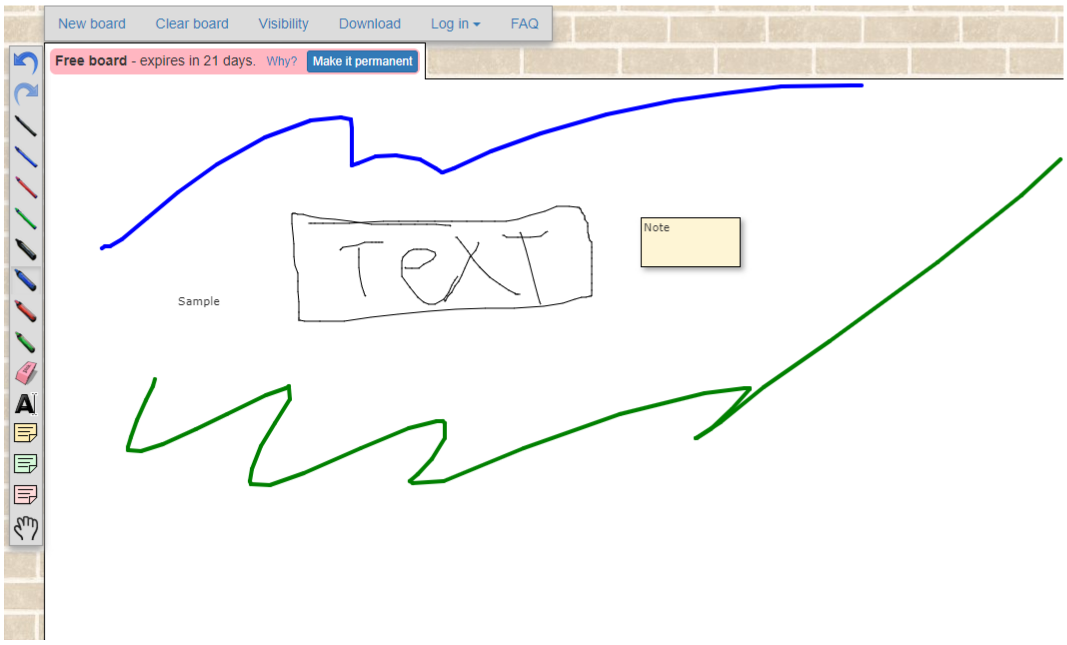


Figure 10: Screenshot of Web Whiteboard

Conclusion:

3 out of the 4 discussed whiteboards used **Canvas** to render the shapes while only one used **SVG**, so I decided to analyze these technologies to see which is more suitable for the Whiteboard (see next sections). I also wanted to know why none of the whiteboards I found used CSS or Flash for rendering.

3.1.2 Cascading Style Sheets (CSS)

CSS is a style sheet language used for describing the presentation of a document written in a markup language like HTML. The name cascading comes from the specified priority scheme to determine which style rule applies if more than one rule matches a particular element.

CSS can be used to create shapes and does it very well[17] [18], however it was not designed for this purpose.

Since CSS was created to style documents and elements, drawing with CSS is basically creating empty HTML elements and styling them so that they *look like* the wanted shapes.

For example: to create a triangle, an element with zero height and zero width is created, the border thickness is set to the height of the triangle, and only one of the border-edges is left in place while the others are deleted[19].

[17] https://www.w3schools.com/howto/howto_css_shapes.asp
[18] https://codepen.io/jaysalvat/pen/HaqBf
[19] https://www.w3schools.com/howto/tryit.asp?filename=tryhow_css_shapes_triangle-up

I decided not to include this technology as a candidate for the Whiteboard. The way that some shapes are created feels ‘hacky’, unintuitive and not developer-friendly.

3.1.3 Flash

Flash is a multimedia software platform created by Adobe [20] and is used for creating animations, desktop, mobile and Internet applications as well as multimedia players. Flash can display text, vector graphics, and raster graphics, it can also capture mouse, keyboard, microphone and camera input.

While searching for more information about this technology, I stumbled upon the fact that Adobe is going to drop Flash support by 2020[21], that is why I decided not to do any further research and to drop this candidate. I do not want to use (soon to be) deprecated technologies because it only means that whoever decides to work on the Whiteboard in the future is going to have to start from scratch.

Another factor for discarding this technology is the fact that it is not natively supported by all browsers. It is only possible to use Flash by installing a plug-in, more specifically, the Adobe Flash Player[22]. Even though some browsers (such as Google Chrome) have an embedded version of the player, it is disabled by default for security reasons. If Flash was used in the Whiteboard, installing Adobe Flash Player will become a prerequisite for using Saros, and it does not make sense to force a dying technology on the users.

3.1.4 SVG

As described in the original specification[4]:

> ”SVG is a language for describing two-dimensional graphics. As a standalone format or when mixed with other XML, it uses the XML syntax. When mixed with HTML5, it uses the HTML5 syntax.[. . .]
>
> SVG drawings can be interactive and dynamic. Animations can be defined and triggered either declaratively (i.e. by embedding SVG animation elements in SVG content) or via scripting.”

Browsers can understand SVG as a part of the HTML document and render the corresponding shapes, browser SVG renderers use the **retained rendering mode**, in which developers issue drawing commands, the browser creates in-memory models in respect to these commands, and then uses these models to display the corresponding shapes on-screen. The retained rendering mode also means the availability of DOM access, which eases the alteration or removal of the shapes.

- Pros:
  - native support from all recent browsers.[23]

[20] https://www.adobe.com/
[21] https://theblog.adobe.com/adobe-flash-update/
[22] https://get.adobe.com/flashplayer/
[23] https://caniuse.com/#feat=svg

  - scalable: the elements can be easily resized.
  - sharp: the displayed shapes are visually appealing with smooth and sharp edges with no jagged lines or blurriness, regardless of size or zoom level.
  - retained rendering: the provided DOM-access is a huge plus, it makes the manipulation of already-created elements an easy and straightforward task.
  - small amount of data needed for rendering complex objects, fewer amounts of data need to be sent across the network while synchronizing.
  - the original Whiteboard uses the SVG standard for properties and attributes, which means the data from the model can be used directly in the browser.
  - support for event handlers: returning back to the Observer Pattern, SVG elements provide a wide variety of events[24], all of which can be used by developers to easily track state-changes on these elements. For example, the 'click' event is triggered when users click on an element, this event can be used to select that element.

- Cons:

  - slower rendering in comparison to Canvas (See 3.1.5), mostly because of the DOM access, since manipulating any element would set a dirty bit on the DOM sub-tree containing that element, thus causing a re-render of the whole sub-tree including elements that did not even change.

    This is not an issue when dealing with a small number of elements (which is the case in the Whiteboard).

3.1.5 HTML Canvas

The Canvas element provides an API with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly. [3]

The Canvas element has a JavaScript API which is the only way to interact with Canvas, it uses the **immediate rendering mode**, in which the developer needs to work out the commands to draw objects, create and maintain the model or scene of what the final output should look like, and specify what needs to be updated. The browser simply executes the drawing commands.

Canvas has no idea of its contents and thus does not provide DOM access for its elements. Since the canvas is bitmap-based, the contents are simply colored pixels.

- Pros:

  - native support from all recent browsers.[25]
  - fast rendering in comparison to SVG, since only the contents of the canvas are updated and no DOM-manipulation whatsoever happens.

[24] `https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/Events`
[25] `https://caniuse.com/#feat=canvas`

  - allows outputting contents as an image, which is a nice feature to have in the Whiteboard.

- Cons:
  - immediate rendering: the lack of DOM access makes the implementation process much harder since everything needs to be created from scratch.
  - resolution dependent, causing low-quality rendering output and jagged/blurry lines when resized.
  - no support for event handlers, forcing developers to manually trigger state-changes, which in turn creates more coupling between the components.

3.2 Deploying the First Prototype

I decided to use SVG, the comparison showed its superiority in this context. The very first prototype was simple and could only draw rectangles, with the same drawing mechanism of the original Whiteboard.

I tested the prototype with Google Chrome and it worked as expected, so I decided to run it in Eclipse's internal browser to see if it would work. After setting up the Eclipse's internal browser to run the prototype, it displayed a blank white page and I was not able to draw any rectangles. After further investigation, I found that the browser has actually loaded the Whiteboard but was not able to render the rectangles using SVG. It also revealed a false assumption I had made regarding the runtime environment of the Whiteboard.

My Assumption: all developers will have a recent and updated browser version, and since Eclipse uses the default system browser internally; all *Saros Users* will have a good browser.

Why it is wrong: Even though Eclipse uses the default system browser, changing the system's default browser does not change it in Eclipse.

In my case, running a Windows 10 machine and Google Chrome as default web browser, it did not work as expected because newer versions of Windows (8 and up) have a version of Internet Explorer left for backward compatibility and it ended up being used in Eclipse.

3.3 Eclipse's internal browser

I have decided to investigate further to know the limits of the internal browser and as it turned out, the version of Internet Explorer used in Eclipse on my machine is IE9, released in 2011.

While online sources[26] list this browser as the first IE that is SVG compatible, the testing revealed a very limited set of available features, which was not enough for the Whiteboard.

I tried swapping the internal browser of Eclipse with a newer, more advanced one, Eclipse, however, limits my choice to Mozilla Firefox[27], which has the needed

[26] https://caniuse.com/#feat=svg
[27] https://www.mozilla.org/

capabilities of rendering SVG, but this also means that having Firefox is a prerequisite for using Saros, that is why I considered this solution to be the very last if nothing else worked.

There has been some work done on creating a Chromium[28] based browser for SWT[29], which I could integrate into Eclipse and use. The problem here is that this browser is still in its earliest versions and very unstable [30] and does not provide any means of communication between Java and JavaScript, rendering it useless in the case of the Whiteboard. Another argument against this browser is that it works in Eclipse only, and would not work in IntelliJ, using it results in an implementation that is not IDE-independent.

An alternative I found was to use an external web browser instead of the integrated. While this would indeed work; it would also create a new set of problems, mainly the management of IPC (Inter Process Communication) between the browser and Eclipse. Furthermore, having an external window defeats the purpose of Saros being an IDE plug-in rather than a standalone software.

Since I found no other solution, I decided to use the default browser in Eclipse and but also find workarounds to mitigate its weaknesses. I found a hack[31] to change the compatibility mode of the internal browser to a higher version, this was, however, system-dependent and required tweaking the registry of Windows, which is a rather unwise thing to do.

In the same thread where I found the hack, there was a mention of another method for changing the compatibility mode, more specifically, using special launch-parameters when starting Eclipse. However, Saros as a plug-in cannot and should not control the launch-parameters of Eclipse.

Searching the open-source community for answers or alternatives I found libraries such as svgweb[32] and dojogfx[33] (with which Sketchbook was created) which provided a fallback for SVG rendering using Flash. Again, the problem with those libraries is the need for an external Flash plug-in to work.

Even though SVG has proven itself to be the most fitting technology, it was unfortunately not compatible, so I had to find an alternative.

3.4 Fabric.js

At the end I choose a library called **Fabric.js**[34] which internally uses Canvas (instead of the intended SVG):

> "Fabric.js is a framework that makes it easy to work with HTML5 canvas element. It is an interactive object model on top of canvas element. It is also an SVG-to-canvas parser."[35]

[28] https://www.chromium.org/Home
[29] https://github.com/maketechnology/chromium.swt
[30] it crashes on right click: https://github.com/maketechnology/chromium.swt/issues/12
[31] https://bugs.eclipse.org/bugs/show_bug.cgi?id=404543
[32] http://code.google.com/p/svgweb/
[33] http://docs.dojocampus.org/dojox/gfx/
[34] http://fabricjs.com/
[35] https://github.com/fabricjs/fabric.js

Even though this library used the Canvas element and thus the immediate rendering mode, it keeps track of its contents and provides DOM access, essentially eliminating the biggest weaknesses the Canvas element has. This DOM access is however limited, there is no support for element hierarchies, meaning no parents or children are allowed, which I had to consider while implementing the new Whiteboard to not cause any problems.

After seeing the demos and skimming through the documentation, Fabric.js appeared to be a very good candidate, so I started working on a second prototype.

4 The Fabric.js Prototype

Re-implementing the first prototype (which, again, could only draw rectangles) with Fabric.js went smoothly, the library provides detailed documentation, many examples and demos, rich functionality and is generally developer-friendly with its Object-Oriented-Programming structure and simple naming conventions.

The library also provides out-of-the-box support for resizing and moving elements around, as well as a free hand drawing mode, which was a huge plus and helped me focus on the next question: how is it possible to synchronize the views in the browser with the `Records` in Java?

4.1 Synchronizing The JavaScript GUI with the Java Model

The first step towards this problem was to understand how the current Whiteboard synchronizes its views with the model so I can build upon it, I found out after inspection that the views work using the aforementioned Observer pattern, which is the standard solution in MVC.

Anytime a `Record` is created, edited or deleted, this `Record` fires an event, which is handled by the corresponding view to render any changes or to create new child-views (if needed). It is worth mentioning that there is no central event management system in the Whiteboard, instead, every `Record` is an independent Subject with the corresponding view being the Observer in respect to the observer pattern.

The first approach that came to mind to eliminate this synchronization problem was to completely move the models to the browser. While this will indeed work, I will have to also re-implement almost the whole Whiteboard, which I considered to be the last solution in case nothing else worked.

Another approach was to duplicate the model data into the browser, this would also introduce the possibility of the Java model and JavaScript model getting out of sync (as in the browser's models contain different information than Java's models), this is most likely to happen when an invalid record or a record-conflict is created.

Synchronization problems such as this one can be solved using a type of consistency watchdogs, which checks periodically to see if both versions of the model are in sync, but this solution would only hide protocol or implementation errors and should only be used as a last resort; thus it is discouraged.

To mitigate synchronization problems, it is possible to implement record conflict resolution in the browser by re-implementing the `SXEController` in JavaScript. This will further complicate the interface between Java and JavaScript to an unreasonable level since the original models are still in Java whereas the duplicate models, the controller and the new view are in JavaScript. Simplifying the interface can be achieved by moving the models to JavaScript, which again means re-implementing everything.

Since trying to keep both models synchronized did not appear to be the right solution, a different approach was required, so I investigated preventing the desynchronization in the first place, rather than fixing it when it happens.

The models can get out of sync in case one them is altered differently than the other one, so I started by finding out what can manipulate the models and cause such problems. After further inspection, I was able to limit model manipulation to two ways:

- User Interactions:
  the changes resulting from these interactions will be transferred from JavaScript to Java, meaning that the only way to desynchronize the models is if JavaScript sent invalid or conflicting records to Java. By taking extra care while developing the new GUI, I could avoid problems here.

- Remote Messages:
  these are 'dangerous' because their validity is unknown.

After analyzing how messages are processed in the Whiteboard, I found out that the `SXEController` fires three types of message events, all of which provide the corresponding message as a parameter:

- `MessageSent`:
  this event is fired after a message is sent and can be ignored since the changes in this message are already applied locally.

- `MessageReceived`:
  this event is fired after processing a received message, firstly, the message is cleaned up of invalid records, the conflicts are resolved, any records depending on other missing records are queued, then the event fires.

- `StateMessageReceived`:
  state messages are sent at the beginning of the session and contain all of the records in the session's host Whiteboard, they are also processed as earlier discussed, the only difference here is that state messages cannot have nor create conflicts.

I decided to test the information provided with these events (especially the `MessageReceived` and `StateMessageReceived` events) to see if they are a valid source of truth, and I found that they contained a clean set of records that indicate exactly the changes made to the Whiteboard.

With all that in mind, I built in JavaScript a simple version of the model that contains just enough information to allow the communication between the environments. This model is only updated through user interactions or filtered messages coming from the `SXEController`. The data is transferred between both environments in JSON (JavaScript Object Notation) format, the serialization is achieved using the GSON[36] library.

However, this solution was not perfect and created a new problem, what if a user opens the Whiteboard after some other user had already drawn something? Since the browser instance does not exist, the fired events are never caught and the information they contain is lost. I considered implementing a message queue to solve this problem, but then again, what if the user does not want to use the Whiteboard at all? the amount of queued messages will explode.

Since the `SXEController` applies all changes in the received messages to the models regardless of the existence of the browser instance, it should be possible to recursively go through all `Records` and update the browser's models as needed.

I tried using the aforementioned GSON library to serialize the `Records` and send them to the browser, this did not work because of the circular relationships between these `Records`; parents have references to their children, children have reference to their parents. These circular relationships were causing infinite recursion and stack overflow errors when trying to convert them to JSON.

After searching for a solution, I found out that is possible to choose which properties to serialize using the `@Expose` annotation, only properties that are annotated or "exposed" will be serialized, others will be ignored. By annotating children references and ignoring parent references no circular relationships were being created and the problem was solved.

With Fabric.js and the underlying synchronization infrastructure, I had created a solid base to build upon.

5 The Last Prototype

At this point, I had answers for all questions regarding the new Whiteboard's runtime environment and the technologies to use, there was still one step missing before I can start implementing, which is the build process.

5.1 The Development Environment and Build Process

As a web developer myself, I had a good idea of state of the art development tools for building web apps, and getting started on this task was straightforward. In this subsection, I will briefly describe the main tools used.

5.1.1 Webpack

"At its core, webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles."[37]

[36]https://github.com/google/gson
[37]https://webpack.js.org/concepts/

Webpack basically creates one JavaScript file out of the whole project thus making the integration of the web application's logic into its respective web document a trivial task.

It is worth noting here that JavaScript code does **not** need compilation in the traditional meaning as in converting the high-level code to assembler code, web browsers can run JavaScript code as it is.

One of the powerful features of Webpack is the ability to choose *loaders*, which allow Webpack to process files in a different way and converts them into valid modules that can be used by the application, one of these loaders is Babel.

5.1.2 Babel

To understand Babel's role, it would be better to explain first:
ECMAScript (or ES): is a trademarked scripting-language specification, it was created to standardize JavaScript[38].

The version I am using in the development of the Whiteboard is ES6 or ES2015, which has a fairly decent support by recent browsers[39]. There are newer versions of this specification but they are not fully supported by most browsers [40] and thus are not recommended.

While browser developers are always trying to integrate any new features or syntax of this specification into their browsers, one should be careful not to make the same mistake I made while implementing the first prototype: "All Saros users will have a good browser", and here is where Babel comes in play.

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backward compatible version of JavaScript in old browsers or environments[41]. Now, of course, Babel will not add an SVG rendering engine to the browser if it does not have one, but it allows development using a newer, cleaner, developer-friendlier syntax, and then compiles this code to a more old-browsers-friendly version.

5.1.3 Other tools and libraries

- React[42]:
  A JavaScript library for building user interfaces, I used it for creating the menu (See 5.2).
- Url-loader:
  converts images to base64-data-URLs and inlines them in JavaScript.
- inlineSource:
  inlines stylesheets and scripts into the HTML document, thus merging all of the Whiteboard dependencies to a single file.

[38]https://en.wikipedia.org/wiki/ECMAScript
[39]http://kangax.github.io/compat-table/es6/
[40]http://kangax.github.io/compat-table/es2016plus/
[41]https://babeljs.io/docs/en/index.html
[42]https://reactjs.org/

5.2 The Menu

I chose to build the menu using React, mainly because of its use in the HTML GUI of Saros [13]. I wanted the Whiteboard to be consistent with other parts of Saros so that any future developers do not feel alienated by the technologies used. Another argument for using React is its popularity, according to a survey done by StackOverflow [43], one of the most known developer communities, React is the most loved technology by front-end developers in 2017 [2].

To ease the development and guarantee extensibility of the menu, I defined a data structure that contains information regarding the elements of the menu, these elements are then created dynamically using React, they are also easily extended just by modifying the aforementioned data structure.

5.2.1 Menu size

It should not be ignored that Saros is an IDE plug-in, and IDEs main purpose is not 'whiteboarding'. Consequently, it should be considered that the Whiteboard might not run in full-screen all the time. In fact, the Whiteboard should run in any window size, which is something I considered while designing the menu; having a too big menu will be a waste of space and will minimize the drawing space or even cover the actual contents of the Whiteboard, having a too small menu causes usability problems.

The solution I decided to implement was a resizable menu, so that users can switch between normal- and mini-mode. I have added later a minimalistic menu that appears next to the cursor when the user clicks the right mouse button for further accessibility and ease of use.

From this point onward adding new tools to the Whiteboard was straightforward, I would implement the new tool, add a new entry in the menu and its done.

6 Conclusion and Outlook

6.1 The State of The New Whiteboard

At the time of writing, the Whiteboard supports drawing rectangles, circles, ellipses, lines and text as well as free hand drawing. Selection mode is also available, allowing the users to move, resize and delete shapes.

Furthermore, two new canvas-related tools are added: increase canvas size and resize canvas to fit all objects, so that users still have enough space to draw on regardless of the window size.

Internally, the changes were minimal, the new whiteboard uses the same Model and Controller as the old one, as a result, both old and new Whiteboards can be used interchangeably. The synchronization between Java models and JavaScript models is ensured, and the new GUI is carefully implemented so that no conflicts arise.

[43] https://stackoverflow.com/

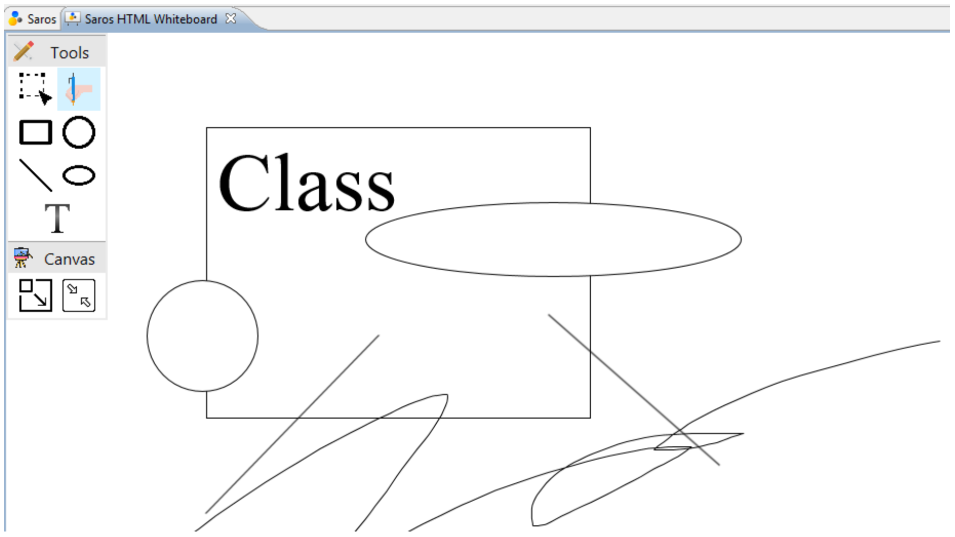


Figure 11: Screenshot of the new HTML-based Whiteboard

6.2 Final Thoughts

In this section I will discuss the decisions I made in the context of this thesis taking into consideration the effects they had on the end result. The reasons for why I made these decisions can be found in their respective (sub)sections.

- Choice of the rendering engine (see 3.1):
  While this decision has indeed changed (see 3.2), I am still convinced that SVG would be the most fitting solution available for the Whiteboard, mainly because of the native support and the internal DOM.

  However, after working with Fabric.js, I learned to appreciate how powerful Canvas actually is, not having an internal structure means that developers have to build it and maintain it themselves, but that also means that the developers are not controlled by what is available and are not bound to any restrictions or limitations, they are free to shape the logic however they find most fitting for the problem in hand. I think this is the reason why most of the discussed whiteboards (See 3.1.1) are created with Canvas.

- Sticking with the default browser for Eclipse (see 3.3):
  I invested a lot of time trying to get SVG to work in Eclipse's browser as well as trying to change Eclipse's internal browser, I did not succeed, but I was able to mitigate the weaknesses of this browser.

- Using Fabric.js as replacement for SVG (see 3.2):
  The library provided a lot of functionality that made the development process easier. There is a chance that I missed some other good library, but I think Fabric.js provides enough and any more functionality would be redundant.

- The development environment (see 5.1):
  Bundling all sources into one HTML document might be regarded as an

‘overkill’, but I think it is necessary to make the integration process of the Whiteboard into different IDEs and runtime environments as easy as possible.

In the context of web apps doing so might be considered sub-optimal, mainly because of the slower downloading and consequently longer wait times, however, the Whiteboard will be loaded and executed locally so loading time is not an issue.

6.3 Incomplete Features

- Path resizing:
  Paths are Fabric.js objects created in free drawing mode, and all Fabric.js objects provide scaling functionality through functions such as `scaleX()` and `scaleY()`, it is however a type of post-processing scaling, the actual dimensions of the object remain the same. It is like zooming in on an image, the image “gets bigger” but its real size never really changes.

  I was able to convert scaling to actual dimension changes in other shapes, but I could not do that for path objects, since they based on commands and not dimensions. Here is an example command:

  Listing 7: Example SVG Path Command

  ```
  M 10 315 L 110 215 A 36 60 0 0 1 150.71 170.29 L 172.55
  152.45 A 30 50 -45 0 1 215.1 109.9 L 315 10
  ```

  I am sure there are special algorithms for solving this problem, but it is not worth the effort. When in need for a bigger path, deletion and new creation is always an option.

6.4 Missing Features

It was first intended as I started working on this thesis to implement the original features of the Whiteboard, and then add new features using the work of Degener [6] as a base. I did not proceed with the latter because of the complexity of some of those features, the focus on graphic tablets as well as the lack of time. Degener listed these functional requirements as well as some non-functional requirements in his thesis [6, p.99-106].

There are however some features that were completely based on the GEF’s `Command` class and did not get re-implemented in the new Whiteboard, which are: undo, redo, copy and paste. Undo and redo being more complicated since they can cause record conflicts and get the models out of sync.

6.5 Conclusion

In this thesis I was able to achieve the goal I had in mind, which is to create an IDE-independent implementation of the Whiteboard, reducing the porting complexity of Saros and taking it one step closer to being independent from Eclipse.

Furthermore, this goal was achieved by re-using already available components from the old whiteboard and avoiding the costly process of re-implementing, even though some functionality had to be sacrificed, nothing important was ignored. The new Whiteboard provides the required infrastructure to implement any missing features as well as add new ones, and serves as a solid base to build upon in case of further development.

References

[1] W3C Document Object Model. https://www.w3.org/DOM/, 2009. [Online; accessed 25-08-2018].

[2] Developer Survey Results 2017. https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-frameworks-libraries-and-other-technologies, 2017. [Online; accessed 16-07-2018].

[3] HTML Standard. https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element, 2018. [Online; accessed 27-06-2018].

[4] SVG specification. https://svgwg.org/svg2-draft/intro.html, 2018. [Online; accessed 27-06-2018].

[5] Christian Cikryt. Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins, 2015. Master's Thesis at the Freie Universität Berlin.

[6] Hendrik Degener. Benutzerzentrierte Entwicklung eines Nutzungskonzepts für das Saros-Whiteboard unter Berücksichtigung der Unterstützung von Grafiktabletts, 2012. Master's Thesis at the Freie Universität Berlin.

[7] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung, 2006. Diploma Thesis at the Freie Universität Berlin.

[8] Martin Fowler and Matthew Foemmel. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, 2006.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[10] Micheal Jurke. Iterative, Prototype-driven Development of a Whiteboard Feature, 2010. Diploma Thesis at the Freie Universität Berlin.

[11] Glenn E Krasner, Stephen T Pope, et al. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[12] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[13] Marius Schidlack. Neuimplementierung und Weiterentwicklung der HTML-GUI von Saros, 2017. Bachelor's Thesis at the Freie Universität Berlin.

[14] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *ACM Trans. Inf. Syst.*, 5(2):147–167, April 1987.

[15] Laurie Williams and Robert Kessler. *Pair programming illuminated.* Addison-Wesley Longman Publishing Co., Inc., 2002.